Analyzing the Impact of the Simulated System on gem5 Performance

Logan McAllister

Abstract

Computer architecture research relies heavily on simulation to model new ideas, but long simulation times can limit the number of experiments performed. Despite this, little work has been done to analyze the performance of these tools. This paper focuses specifically on the gem5 simulator, using CPU profiling to scrutinize its execution when simulating different kinds of systems. Three main categories of experiments were conducted, profiling how gem5 runs when modeling different instruction set architectures (ISAs), cache configurations, and simulated workloads. This performance data is valuable both for developers of gem5, to indicate the areas of the simulator that run the least efficiently, and for users, to construct simulations that run more quickly. Profiling revealed that the efficiency of the different ISAs depends heavily on other system components, with ARM systems generally running the least efficiently. Experimental workload impacted performance relatively little across different benchmarks, particularly when compared to the differences in performance of the benchmarks run straight on the host system. Finally, larger caches reduced runtime by up to 39%, but primarily through the execution of fewer instructions rather than higher efficiency.

Contents

1. Introduction

Computer architecture simulation is an important step in processor design. It provides a significantly cheaper, less rigid testing ground than field programmable gate arrays (FPGAs), while still giving valuable performance data. With physical limitations on clock speed and the size of transistors blocking the traditional paths of computational advancement, architecture simulation has become more important than ever [1]. Moore's Law [2], which has reliably predicted that the number of transistors on an integrated circuit doubles roughly every two years, can no longer be sustained in the traditional sense. Advancement in hardware now relies primarily on specialization, and architecture simulators provide a place to begin modeling these specialized devices [3][4].

One of the main difficulties facing the field of architecture simulation is the duration of experiments. Typically, the more accurate a simulator's output is, the longer it takes to generate. Among the most widely used of these simulators is gem5 [5]. Gem5 provides detailed microarchitectural data, but with significant time costs. It runs at speeds substantially slower than a native system, with workloads that would normally be measured in seconds or minutes taking hours instead. For example, simulating a Linux boot on an out-of-order x86 processor model takes a minimum of three hours.

This paper takes a look at how gem5 runs, using profiling to analyze its performance with the goal of revealing the bottlenecks that are faced, and what qualities of the simulated system influence performance the most. Even small improvements in the efficiency of gem5 will have a large impact over the course of a full simulation, and this work reveals some areas with the most potential to make adjustments.

1.1. gem5

The gem5 simulator is an open source tool for modeling highly configurable computer systems. Gem5 allows for full system simulation, meaning that a simulated environment can be built with processor cores, caches, memory, and I/O devices, all of which are modular and can be

mostly interchanged without impacting the rest of the system [5]. Simulations are configured with Python scripts, where Python objects provide an abstraction to the C++ code driving the simulation itself. At a high level, gem5 is driven by a system of events. Each component modeled in the simulation can introduce events (functions) to a central event queue where they're scheduled based on timing, so the simulation can jump to the time of the next event without having to simulate each individual clock cycle.

One of the main ways that gem5 stands out from other architecture simulators is the ISA system. gem5 allows simulated processors to run x86, ARM, RISC-V, as well as other less common instruction sets, which all work with any of the processor models offered. The system relies on a domain specific ISA language, with ISA code fed through a specialized ISA parser to generate C++ code for a model of the decoder and execution unit. The x86 implementation is particularly detailed, with instructions broken down into micro-operations that are translated to functions which change processor state appropriately (see fig. 1 and section 1.4 for more details on micro-operations).

gem5 provides three main CPU models, and one special model. All of the main models are architecture agnostic, meaning they can use any of the available ISAs.

- **SimpleCPU:** A linear, non-pipelined CPU implementation that runs quicker than the others at the cost of detail. It's useful for memory focused studies, or any others where the specifics of processor timing don't matter. This CPU model comes in two forms, Atomic, where every instruction takes one cycle no matter what, and Timing, where reads/writes are modeled with accurate timing, but every other type of instruction takes one cycle.
- **MinorCPU:** An in-order, pipelined processor that allows for experiments on pipeline structure and provides much more detailed timing information.
- **O3CPU:** An out-of-order, pipelined processor that most closely resembles a real-world system. This model runs the slowest of the three.
- **KVM CPU:** A special model that allows instructions to run directly on the host processor when simulating a system with the same ISA as the host [6]. This speeds up execution massively, but doesn't provide any timing data, making it primarily useful for fast-forwarding a simulation to a region of interest.

Caches in gem5 are handled mostly through the simulator's Ruby subsystem [7]. This system models cache coherence policies for different styles of caches, ensuring fidelity. Beyond caches, the simulator contains detailed models of different DRAM controllers for memory in a system. For simulated workloads, gem5 provides a library of pre-compiled resources for each offered instruction set.

1.2. Other Simulators – Why gem5

There are two main categories of computer architecture simulators. Functional simulators implement only the general architecture, emulating the target ISA without tracking microarchitectural details. They are typically much faster, but have limited use cases in architecture research. The alternative to these are timing simulators, which can provide detailed data on how the target system runs. Some timing simulators are cycle-accurate, with systems implemented at the register transfer level (RTL) and simulating each individual clock cycle. Others are event-driven, tracking progress through larger events rather than individual cycles. gem5 is a cycle-level event-driven simulator, meaning that at its core it's driven by events rather than cycles, but it still claims to provide nearly cycle-accurate data [5].

Simulators also vary in how much of a computer system they can model, and what inputs they can take. gem5 has the capacity to model a full computer system, and runs real binaries as workloads. Other similar event-based simulators are explored in detail in the survey by Akram and Sawalha [8]. Each of the other simulators examined have some weaknesses relative to gem5 for the purposes of this study. MARSSx86 [9], PTLSim [10] and ZSim [11] are limited to only x86-based target systems, closing off a key avenue for exploration. Multi2Sim [12] and Sniper [13] both support more diverse ISAs, but lack the ability to run full-system workloads such as Linux boots. All of these simulators have areas in which they excel, but none are nearly as configurable as gem5. Combine this with the wide usage of gem5 in academic and industry research [5], and its importance is clear.

1.3 perf

CPU profiling is a method of performance analysis that evaluates the execution of code on the processor [14]. This is carried out using hardware based performance monitoring counters (PMCs) to track specific processor events during execution. These PMCs are registers which increment whenever the chosen event is encountered, controlled by a performance monitoring unit (PMU) with separate registers that determine which events are collected.

Linux's perf profiler, often called perf_events, uses this PMU for CPU profiling [14]. Perf provides a generalized environment for performance analysis across different systems, opposed to manufacturer specific tools like Intel's VTune [15] or AMD's uProf [16]. perf also provides support for the Top-Down profiling methodology (described in the next section) through specialized event groups. perf has a wide variety of profiling options and commands, but this work primarily used perf stat, which simply collects the number of times each event occurred.

A primary issue in CPU profiling is the lack of sufficient PMCs to track every relevant hardware event. This often leads to multiplexing of these counters, where data on certain events is only collected at certain times during execution. Perf uses round robin scheduling for this multiplexing, then estimates the results as if every event was tracked fully [17]. This can lead to inaccuracies in profiling data.

1.4 Top-Down Analysis

Top-Down Microarchitectural Analysis is a profiling methodology that attempts to simplify the process to give clearer information about performance bottlenecks [18]. It requires only a few additional events in the PMU, and has been widely adopted by Intel, with support built into all of their processors released after the paper was published. The experiments in this paper were performed on an Intel Xeon-based server (details in section 3 Table I) following this process.

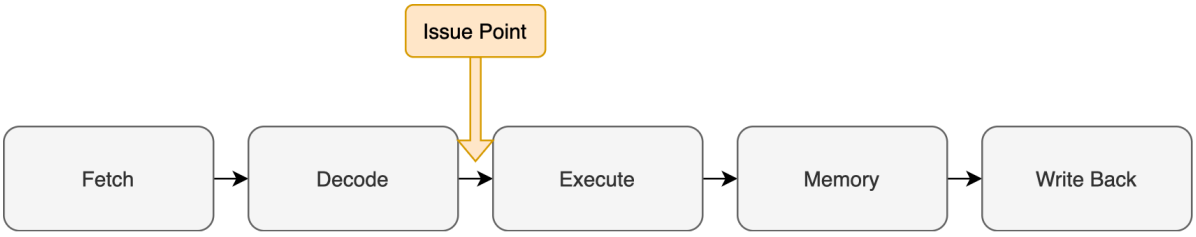

Fig. 1: Basic processor pipeline with uop issue point

Intel machines use the x86 instruction set. x86 processors are complex instruction set computers (CISC), meaning that individual instructions have to be decoded and broken down further before being executed. These sub-instruction level commands are referred to as micro-operations (uops), and the "frontend" of an x86 processor involves both fetching instructions from memory, and this process of decoding them. Top-down analysis creates a hierarchy of execution, focusing on the point in the pipeline where uops are issued from the uop queue to an execution unit (see issue point in fig. 1), moving from the frontend of the pipeline to the backend. It categorizes pipeline slots at the most basic level into four categories: frontend bound, backend bound, bad speculation, and retiring (fig. 2).

- **Frontend Bound:** No uop was issued because none was available to be, due to a stall in the frontend of the pipeline.
- **Backend Bound:** No uop was issued because of a lack of available execution units.
- **Bad Speculation:** Either a uop was issued but on a branch that was not taken, or was not issued because of a stall from previous mis-speculation.
- **Retiring:** A uop was issued that retired, writing to either a register or memory. Retiring cycles are the goal, as they are the only cycles where meaningful work is accomplished. Retiring percentage directly determines instructions-per-cycle (IPC), which indicates how many instructions are executed per clock cycle across the machine, and is a general indicator of efficiency.

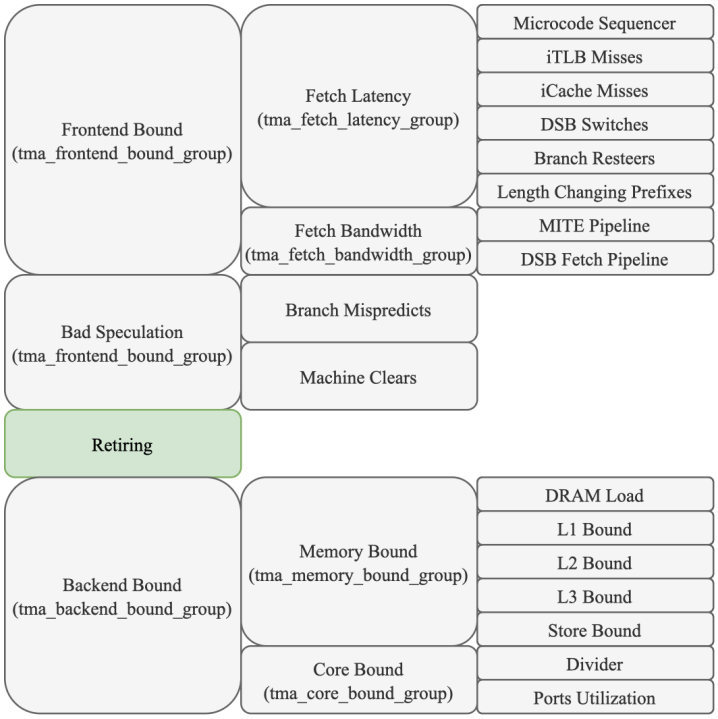

Fig. 2: Top-down Hierarchy (based on diagram found in [18])

After running this top-level analysis (the `TopdownL1` event group on Intel chips), the next step in the procedure is drilling down into the details of the categories (except retiring), which made up a significant portion of execution time. For example, if a process is primarily frontend bound, `tma_frontend_bound_group` can be collected next, indicating whether the stalls are coming from fetch latency (usually instruction cache, iTLB), or fetch bandwidth (uop decoder). These categories can then be expanded further until a specific bottleneck is identified. The Top-Down methodology allows for these bottlenecks to be found without having to guess at which events to collect, and provides a straightforward, well supported process to do so.

2. Previous Work

Gem5 and other architecture simulators play an extremely important role in the development process, but limited work has been done on profiling their performance. The paper from Umeike et al. [19] breaks down how the host platform affects gem5 performance, using the same top-down approach. They were able to isolate certain factors in the host system that had a large impact on the execution of the simulator, specifically finding that a larger L1 cache size drastically improves performance. Their paper was the primary motivator for this work, providing a baseline to compare to and expand upon. They profiled gem5 running Linux boots and PARSEC benchmarks [20] on simulated ARM systems using all three main processor models that gem5 offers (section 1.1). They mention that they also changed the size of memory, and the number of processor cores, but stuck with one setup per processor model. This paper instead turns the focus inward, looking at how the details of the simulated system, rather than the host system, impact performance.

More work has been done on verifying the accuracy of architecture simulators. Two works by Akram and Sawalha [8], [21] examine the performance of several different simulators, focusing primarily on correctness, looking mostly at runtime to compare efficiency. They also provide a detailed survey of the field as a whole and help in defining the key categories of architecture simulators mentioned in section 1.2. Another survey by Hwang et al. [22] compares CPU and memory simulators while also looking at compiler optimizations to enable better performance within them.

3. Methodology

All experiments were run on a server with Intel Xeon cores (table I). While experiments were conducted, no other user interacted with the server, and the gem5 processes were the only non-kernel processes running outside of the shell, and occasional commands to monitor progress.

TABLE I: Host system details

Component	Value
Processor	Intel Xeon CPU E5-2640 v4
Freq	2.40GHz (3.4GHz max)
OS	Rocky Linux 9.5
Cores	20C/40T
Memory	128GiB DDR4
L1 (per core)	32KiB (D) + 32KiB (I)
L2 (per core)	256KiB
L3 (per node)	25MiB
Page Size	4KiB

3.1. Top-down Metrics Collected for gem5

To gather experimental data, the process of Top-down microarchitectural analysis was performed using Linux's perf profiler. This began with the `TopdownL1` event group, which breaks cycles down as frontend bound, backend bound, bad speculation, or retiring (section 1.4). For gem5, simulations were typically primarily frontend bound and backend bound, meaning that the `tma_frontend_bound_group` and `tma_backend_bound_group` were collected next. For all of the experiments conducted by this work, the `tma_fetch_latency_group` was also collected, and with these four a relatively complete picture of execution time could be

created (see fig. 2 for a breakdown of each). Because of the limited PMCs (see section 1.3), all of these event groups could not be collected on the same trial. This meant two things, first that separate trials had to be run for each top-down category, and second that different levels in the hierarchy couldn't be directly compared on the same trial. This breakup multiplied the number of overall trials needed for each experimental configuration by four, which created significant time constraints given gem5's already long runtime.

3.2. Exploration and Timeline

This work began with an exploration of gem5's codebase. Before the goal of this project was profiling gem5, the original intention was to test an implementation of a writable control store within gem5, and to see if different instruction sets could be emulated through microprogramming (explanation below).

Some background is required for this to make sense. In order for a processor to function, it needs a *control unit*. There are two main implementations of the control unit: hard-wired, where the order of the pipeline and the execution of micro-operations (see section 1.4) are manually configured in circuitry, and *microprogrammed*, where these details are instead implemented through a language of *microinstructions* [23]. A microinstruction is a special type of instruction word where the individual bits map directly to control lines that determine what the processor does in a given cycle, so reading a microinstruction is the same as executing it. These microinstructions are used in a small section of memory within the control unit called the *control store*. The contents of the control store determine the actual instruction set of the processor, where each instruction is represented by a sequence of microinstructions which together perform the function of the instruction. Executing an instruction simply involves reading each of this sequence of microinstructions in order.

Microprogramming is a relatively old idea, and is no longer widely used in modern processors [23]. That said, it provides a unique level of flexibility that can be useful for certain niche applications, such as emulation. The process of dynamic user microprogramming [24] relies on a *writable control store*, where the control store is read-write instead of read-only, which allows the instruction set to be modified after the processor is already created. This work

originally intended to implement this concept in a way that would allow for fully interchangeable instruction sets, and to test it using gem5. There were several flaws with this idea, the most important being that gem5 has no built-in support for dynamic microprogramming.

As mentioned earlier, the first step in this process was to explore gem5's codebase, specifically the ISA implementations. The ISA system is based on a domain-specific ISA description language, which is used to rigorously define an instruction set. This ISA code is then fed through a special *ISA parser* which generates C++ code for a decoder and execution unit based on it. The x86 ISA code does include micro-operations, and even a class called `MicrocodeRom`, but their implementation works in a way that isn't compatible with a writable control store. Fig. 3 shows some of the code for this class from two files, `rom.isa`, which is in gem5's ISA description language [25], and `microcode_rom.hh`. This code demonstrates that in gem5, microcode ROM is implemented as an array of type `GenFunc`, which is a typedef representing a pointer to a function that both takes and returns a pointer to an instruction object. To put it simply, gem5 implements microcode memory as an array of function pointers, which all point to a function that executes a specific micro-operation.

```
C/C++
X86ISA::MicrocodeRom::MicrocodeRom()
{
    using namespace rom_labels;
    genFuncs = new GenFunc[numMicroops];
    %(alloc_generators)s;
}

typedef StaticInstPtr (*GenFunc)(StaticInstPtr);
```

Fig. 3: Code from gem5's x86 ISA implementation

All of this code is fed to gem5's ISA parser, as well as definitions for each micro-operation, and the sequence of micro-operations to execute for each instruction. The actual microcode is implemented as a Python string which the parser then converts into C++ code. Fig.

4 shows an example of this for two add instructions, the first between two registers and the second between a register and an immediate. The main problem with this is that once gem5 is compiled, the control store becomes fixed, as it isn't an actual model of memory which can be converted from ROM to RAM. This made the previous goal unfeasible, and required a new direction.

```python
microcode = """
def macroop ADD_R_R
{
    add reg, reg, regm, flags=(OF,SF,ZF,AF,PF,CF)
};

def macroop ADD_R_I
{
    limm t1, imm
    add reg, reg, t1, flags=(OF,SF,ZF,AF,PF,CF)
};
"""
```

Fig. 4: x86 addition instructions defined by microcode

The decision was made to shift toward analyzing gem5's performance, specifically through profiling. Before experiments could be conducted, setup was required to build gem5 properly and create scripts that could dynamically swap out system details between experiments. This setup is found in section 3.3. Following this, the next step was to determine how running experiments in parallel impacted profiling data, which is described in section 3.4. This was done with the hopes of speeding up data collection, as time constraints played a significant role in the decision making process for this project after the late change in topic. Once this was completed, decisions were made about which configurations to test, and what order to test them in.

Three major details of gem5's simulated systems were tested: ISAs, workloads, and cache configurations. The ISA implementations were profiled first, as that was the area of focus for earlier exploration. It was also a natural target for performance profiling given the stark differences between the ISAs. More in-depth procedure for these tests is found in section 4.1. After the preliminary ISA experiments were completed, the next trials focused on how different simulated workloads impacted gem5's performance. The procedure for these experiments can be found in section 5.1. Finally, simulations with different cache configurations were profiled to determine how they affect gem5's performance, with the procedure found in section 6.1

3.3. Setup

Experiments were carried out through the use of two Python scripts, which will be referred to as `run_script` and `batch_script`. `run_script` is a gem5 configuration script that takes command line arguments specifying the details of the system, and the workload to be run. The script constructs a full system gem5 simulation based on these details, and executes it. Gem5 requires ISAs and cache configurations to be specified at compile time, meaning that in order for this script to work for every experiment, a build of gem5 had to be created including all of these. This is one of the preconfigured build options in gem5, with the name "ALL", and can be compiled with: `scons build/ALL/gem5.opt -j {cpus}`.

`batch_script` runs a batch of experiments from an input file. The input file format puts one experiment on each line, detailing all of the specs required to build the system in run_script, and the perf event group to be collected. The example below shows an input file which would run two experiments on ARM O3 systems collecting two different top-down perf metrics.

```
ARM O3 2GiB 32KiB:256KiB 4 boot-exit TopdownL1
ARM O3 2GiB 32KiB:256KiB 4 boot-exit tma_frontend_bound_group
```

`batch_script` uses Python's `os` module [26] to run each experiment as a subprocess, and handles cleanup after each trial. As will be discussed in section 3.4, two trials run in parallel on separate processor nodes. This is accomplished using the `taskset` command [27] to lock

the core affinity of each process to cores on separate nodes. For cleanup between trials, batch_script clears the server's caches of previous experiment data by echoing 3 to /proc/sys/vm/drop_caches [28], then sleeps for five minutes before starting the next pair. This is done to prevent any leftover data from impacting subsequent experiments. In order to run experiments on the server without staying logged in via ssh, the script was run using the Linux `nohup` command [29], which makes a process ignore the hang up signal sent on logout.

3.4. Parallel Trials

Parallelism can be a powerful force in computing, allowing programs to run much faster by splitting work among cores. When doing profiling work, however, parallelism introduces a number of complications. Given the long runtime of gem5 simulations, preliminary experiments were performed to determine how (if at all) running trials in parallel impacts data. It's important to note that this was explored relatively briefly given the limited timeframe of this project, and in order for real conclusions to be drawn, a much more detailed study would need to be conducted. This section is not meant to show anything beyond the rationale behind the number of experiments run simultaneously for later results.

TABLE II: Averages of top-down level 1 metrics on parallel trials

Metric	Sequential	2 Parallel*	2 Parallel	5 Parallel	10 Parallel	15 Parallel
backend_bound	8.14	8.37	8.19	9.12	9.37	9.66
bad_speculation	2.7	2.67	2.64	2.71	2.71	2.78
frontend_bound	43.61	43.51	43.75	42.93	43.04	42.22
retiring	45.52	45.47	45.43	45.21	44.91	45.34
Number of trials	10	10	10	10	20	30

Initial data indicated that any amount of parallel trials could cause changes to profiling data. Table II shows the averages from a sequence of experiments simulating a simple system with a single-core TimingCPU, collecting TopdownL1 data. The category marked with the *

represents data collected when two trials ran in parallel while restricted to processor cores on separate nodes. Between 10 and 30 trials were conducted per category, with a different number running simultaneously. As the number of parallel trials increases, the trials become slightly more backend bound, and slightly less frontend bound (see section 1.4). The variance in each top-down category between each trial also increases substantially with almost any amount of parallelism (fig. 5). The most obvious reason for this is the cache configuration of the Intel Xeon chips. The L1 and L2 caches are per-core, but the L3 cache is shared among all cores on a node. This means that, on the server used by this paper, only two L3 caches exist. This causes the individual trials to affect others running at the same time, impacting results to an extent that hurts the validity of the collected data.

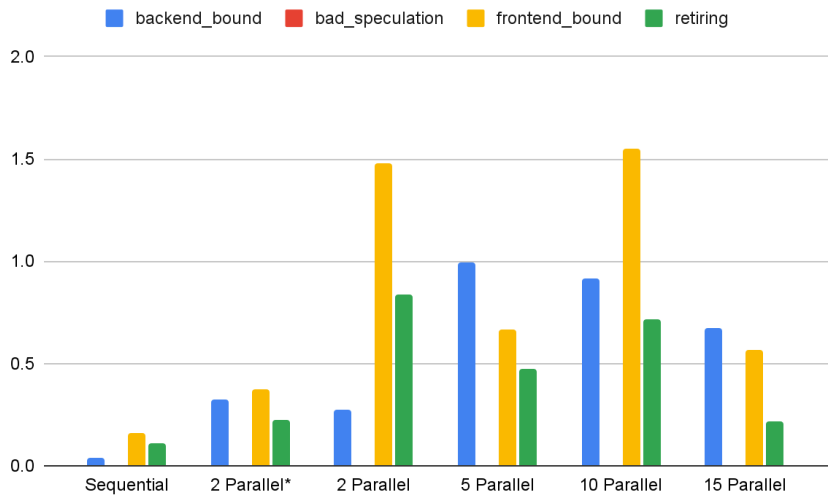

Fig. 5: Variance in top-down categories between parallel trials

Despite these issues, a method was found to collect good data while running two trials together. Using the taskset command, it's possible to set the core affinity of both processes in a way that each can only run on cores of separate nodes. This was also used to avoid a certain core and associated hyperthread that were found early on to give vastly different profiling data from the rest, and run for much longer (see Appendix A). These trials are notated as "2 parallel*" in Table II and Fig. 5, and show that the averages and variance are both much more in line with sequential experiments.

4. ISA Experiments

4.1. Procedure

The first major category of experiments performed were on gem5's ISA implementations, specifically the x86, ARM, and RISC-V ISAs. Profiling data was collected on two simulated systems, referred to as basic and complex, which are detailed in Table III. The basic system has a single-core TimingCPU, and represents an overly simplified system that does not provide detailed data, but runs fast. The complex system has a four-core O3CPU, modeling a much more realistic system at the cost of speed. For the basic model, 10 trials ran for each of the three ISAs. As explained in section 3.1, four top-down metric groups needed to be collected for each experimental configuration, for a total of 120 trials. For the complex system, time constraints limited collection to five trials per ISA and metric group, for a total of 60 trials.

The experimental workload for each of these trials was an Ubuntu 24.04 Linux boot ({isa}-ubuntu-24.04-boot-no-systemd), which will be referred to in the future as *boot-exit*, from gem5's resources system [30]. Given the large differences between the 3 examined ISAs, it wasn't possible to run exactly identical trials, but these similar Linux boots gave as close of an approximation as possible.

TABLE III: System details for ISA experiments

Component	**Basic**	**Complex**
Processor Model	Timing	O3
Memory	1GiB	2GiB
Cores	1	4
L1	32KiB	32KiB
L2	256KiB	256KiB

4.2. Data and Analysis

TABLE IV: ISA top-down metrics, as percentages of cycles

		Simple			Complex		
Event Group	**Metric**	**x86**	**RISC-V**	**ARM**	**x86**	**RISC-V**	**ARM**
TopdownL1	Backend Bound	8.4	7.67	9.49	15.44	14.64	13.62
	Bad Speculation	2.7	2.88	3.42	4.18	5.28	5.58
	Frontend Bound	43.5	44.09	45.07	49.24	47.24	50.74
	Retiring	45.5	45.36	42.04	31.12	32.84	30.06
Frontend Bound	Fetch Bandwidth	21.03	20.73	18.75	12.46	14.02	13.8
	Fetch Latency	22.36	22.9	26.23	36.92	33.66	35.86
Backend Bound	Core Bound	4.58	5.02	5.03	6.48	6.3	6.1
	Memory Bound	3.64	2.64	4.49	9.06	8.36	8.32
Fetch Latency	Branch Resteers	4.8	6.15	6.97	12.62	11.16	11.84
	DSB Switches	1.81	2.19	2.12	1.98	1.98	1.84
	iCache Misses	9.15	9.74	12.19	19.7	17.48	18.6
	iTLB Misses	9.79	9.42	9.94	6.66	6.8	7
	LCP	0.4	0.9	0.2	0.1	0.4	0.1
	MS Switches	1.5	1.4	1.5	1.2	1.66	1.5
General (from L1)	Time Elapsed (s)	3036	2500	2130	12,382	6764	4708
	uops (trillion)	19.35	15.92	12.67	57.37	33.72	21.77
	Number of trials	10	10	10	5	5	5

Table IV shows the averages of multiple levels of top-down metrics, as well as total time elapsed, number of uops retired, and number of trials per level. From a glance, it's clear that

different trends exist between the two models. Time constraints limited the number of experiments that could be performed, so analysis of which changes between the systems impact the different ISAs in what way will be left to future studies.

The most important metric in these experiments is retiring cycles (highlighted in Table IV), as they indicate what percentage of clock cycles the processor was performing useful work. On both systems, ARM runs the least efficiently of the three. This primarily comes from the processor frontend, and specifically from instruction cache misses. Gem5's ISA system relies heavily on generated code (see section 3.2) to construct a model for the decoder and execution unit, and Table V shows the sizes of these generated files from each ISA. ARM is noticeably larger than the others, which matches with observations of the high portion of cache misses it experiences. Interestingly, x86 is the smallest of the three, despite it being the only one to implement micro-operations. x86 also suffers from a high percentage of cycles bound by instruction cache misses, particularly on the complex system. The smaller code size likely stems from the fact that the execution unit only has to implement functions for micro-operations, rather than every single instruction, which is further backed by the much smaller size of its execution unit relative to the others. This could also be the cause for the worse instruction cache performance, as each instruction called within the gem5 simulation would need to fetch code from multiple microinstruction functions.

Generated code naturally tends to have areas with potential for optimization, and one concept for a future study would be to conduct a frequency analysis of instructions run within gem5, and experiment on ordering in the generated code to take better advantage of locality. This will be discussed further in section 7.2.

TABLE V: Sum of the sizes of generated ISA files

	X86	**RISC-V**	**ARM**
Decode	83MiB	104MiB	142MiB
Execute	14MiB	76MiB	183MiB
Total	97MiB	180MiB	324MiB

5. Workload Experiments

5.1. Procedure

This category of experiments focused on what the simulated system was running: the workload. These experiments ran on an X86 processor matching the specs of the complex system from the ISA experiments (Table III). The PARSEC benchmark suite [20] provides different programs designed to stress multi-core systems in unique ways. PARSEC benchmarks were used for these trials for three main reasons. First, they're extremely diverse in *working set* (the amount of memory consistently used by the process), *locality* (the frequency with which the same section of memory or those nearby are accessed), and computational intensity. Second, each benchmark comes in different sizes, some designed specifically for use within simulators. All experiments in this paper use the `simsmall` size. Third, gem5 has built in support for these benchmarks through the `x86-parsec` resource, which has been tested on Ubuntu 18.04 systems.

The following three benchmarks from PARSEC were used for these experiments:

- **blackscholes:** Uses the Black-Scholes differential equation to calculate prices for European options. Chosen for frequency of floating point calculations, and relatively small working set.

- **canneal:** Cache-aware simulated annealing for chip design. Unbounded working set and lots of memory writes. Chosen primarily for stress it puts on simulated caches.

- **fluidanimate:** Simulates and animates the flow of a fluid based on physics. Large working set and intensive both in memory usage and calculations. Chosen for general system-wide stress.

This benchmark suite needs to run on a Linux system, so Ubuntu 18.04 was booted within the simulation before the benchmarks themselves ran. In order to focus profiling

specifically on the benchmarks, and to save time, the boots were performed with gem5's KVM CPU (see section 1.2) to fast forward to the region of interest, before switching to the O3CPU. Five trials were performed for each of these benchmarks, for each of the four top-down metric groups collected, for a total of 60 trials. The benchmarks were also run directly on the host system, not through gem5, to provide a comparison.

5.2. Data and Analysis

TABLE VI: Workload top-down metrics as percentage of cycles

		gem5			Host		
Group	**Metric**	**blackscholes**	**canneal**	**fluidanimate**	**blackscholes**	**canneal**	**fluidanimate**
TopdownL1	Backend Bound	21.06	19.2	20.32	31.46	44.34	41.02
	Bad Speculation	6.48	6.12	6.94	6.44	7.58	6.6
	Frontend Bound	45.14	46.62	46.5	31.54	16	9
	Retiring	27.34	28.08	26.26	30.58	32.1	43.32
Frontend Bound	Fetch Bandwidth	11.12	12.1	11.04	8.64	5.56	2.9
	Fetch Latency	33.98	33.94	35.6	22.24	10.26	6.2
Backend Bound	Core Bound	8.14	7.74	7.58	11.72	16.8	25.04
	Memory Bound	13.06	11.9	12.86	18.6	28.5	15.9
Fetch Latency	Branch Resteers	12.58	12.28	13.62	13.54	5.78	5.76
	DSB Switches	1.94	1.7	2.1	1.74	2.42	0.44
	iCache Misses	18.1	17.34	18.8	9.88	2.22	1.82
	iTLB Misses	5.48	4.92	4.9	1.74	0.4	0.42
	LCP	0.1	0.1	0.1	0.1	0	0
	MS Switches	1.46	1.8	1.4	3.02	1.38	1.12
General (from L1)	Time Elapsed (s)	1856	6997	17,062	0.38	1.37	0.76
	uops (trillion)	7.42	30.40	72.85	0.00137	0.00765	0.00982

Simulations where the same system ran different PARSEC benchmarks, shown in Table VI, performed remarkably similarly. These benchmarks were intended to each tax the system in different ways, but instead seemed to make relatively little impact on how efficiently it ran. The

relative size of the benchmarks in comparison to the size of the simulator itself made the work performed within the simulation less impactful than it otherwise would have been. This is highlighted further by Fig. 6, which shows `TopdownL1` data from each benchmark both within gem5 and directly on the server. On the server, fluidanimate performed significantly better than the other two, and actually took less time to run than canneal. Within gem5, fluidanimate performed the worst of the three, and ran for the longest by far. This difference comes both from performance and from the amount of work done.

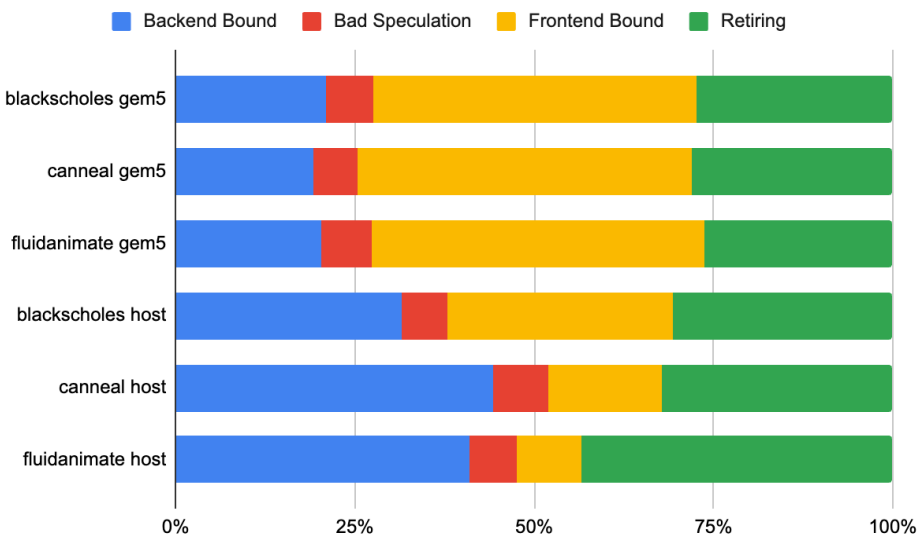

Fig. 6: TopdownL1 workload data including boot-exit

Within gem5, fluidanimate caused the simulator to perform nearly 2.5x as many total micro-operations as canneal, while on the server, it performed only 1.28x as many. This suggests that some aspect of fluidanime causes it to require substantially more work to model than the other two. The most likely cause for this is the benchmark's large working set, which would lead to frequent cache misses within the simulation, causing lots of accesses to the DRAM model. As section 6.2 will show, accessing memory from the cache requires substantially less computation than from memory. This doesn't reflect in top-down performance metrics as the simulation isn't actually running less efficiently, it's just performing more work.

Another interesting trend is the poor performance of the blackscholes benchmark on the server, particularly in fetch latency. Fig. 7 shows that blackscholes performed substantially worse

than the other two benchmarks in these metrics. It was particularly bound by instruction cache and instruction TLB misses. Despite this massive difference, it didn't translate at all into the simulation, as gem5 didn't perform noticeably worse in either of these metrics while running this benchmark than the other two.

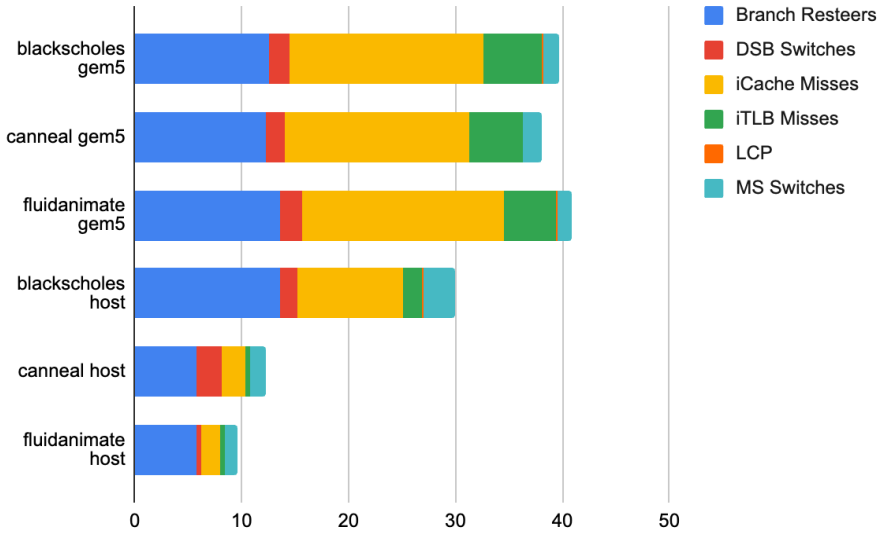

Fig. 7: Fetch latency metrics

6. Cache Experiments

6.1. Procedure

The final category of experiments performed were on configurations of gem5's caches. Two cache models were used, `MESITwoLevelCacheHierarchy` and `MESIThreeLevelCacheHierarchy`, to model two and three-level caches respectively. Two sets of sizes for each cache hierarchy were tested, shown in Table VII.

TABLE VII: Cache configurations

Cache Level	Config 1	Config 2	Config 3	Config 4
L1	32KiB	64KiB	32KiB	64KiB
L2	256KiB	512KiB	256KiB	512KiB
L3	none	none	2MiB	4MiB

Experiments were performed on an x86 system matching the complex system from section 4.1 (Table III). These simulations ran the PARSEC blackscholes benchmark (see section 3.6). This was chosen for two main reasons, first because it performs a large number of reads to fetch options data from memory [20], and second because of its relatively low runtime compared to the other choices. Given the four different configurations, and that four sets of trials had to be run for each to get all of the relevant top-down metrics, 80 trials were conducted in total to get five for each category.

6.2. Data and Analysis

TABLE VIII: Cache top-down and other metrics

Event Group	Metric	Config 1	Config 2	Config 3	Config 4
TopdownL1	Backend Bound	21.06	21.5	27.4	26.66
	Bad Speculation	6.48	6.58	6.24	6.26
	Frontend Bound	45.14	45.3	39.26	39.64
	Retiring	27.34	26.6	27.04	27.48
Frontend Bound	Fetch Bandwidth	11.12	10.86	9.56	9.26
	Fetch Latency	33.98	34.3	30.68	30.5
Backend Bound	Core Bound	8.14	8	9.8	9.8
	Memory Bound	13.06	12.72	16.54	16.44
Fetch Latency	Branch Resteers	12.58	12.7	10.64	11.22
	DSB Switches	1.94	1.94	2.06	2.06
	iCache Misses	18.1	18.16	16.66	17.14
	iTLB Misses	5.48	5.62	5.3	5.44
	LCP	0.1	0.1	0.1	0.1
	MS Switches	1.46	1.4	1.2	1.22
General (from L1)	Time Elapsed (s)	1856	1660	1129	1138
	uops (trillion)	7.42	6.43	4.22	4.22

Table VIII shows top-down data from trials on the four cache configurations. The most noteworthy results from these experiments actually come from the time elapsed and number of uops retired, rather than the top-down data. Generally speaking, the larger the overall size of the cache, the faster the simulation ran. More experiments need to be performed to analyze where this hits a point of diminishing returns, and to analyze how the program that the simulation is

running impacts it. The optimal cache size for performance likely depends directly on the working set of the workload. This difference in timing likely comes from gem5's implementation of DRAM, as data indicates that memory accesses require significantly more instructions than cache hits to model.

Interestingly, despite the substantially lower runtimes, larger caches do not make the simulator run that much more efficiently. Configurations three and four with these larger, three level caches were substantially less frontend bound than the others, but were correspondingly more backend bound. This ultimately levels out, as shown in Fig. 8. Breaking down the frontend bound category further, two-level cache experiments were bound more by both fetch bandwidth and latency, suggesting that memory accesses have a larger instruction cache and uop cache miss rate. Looking at backend bound cycles, experiments on three-level caches tended to be both more core and memory bound than two-level, which could imply that cache accesses in general have these bottlenecks in comparison to memory.

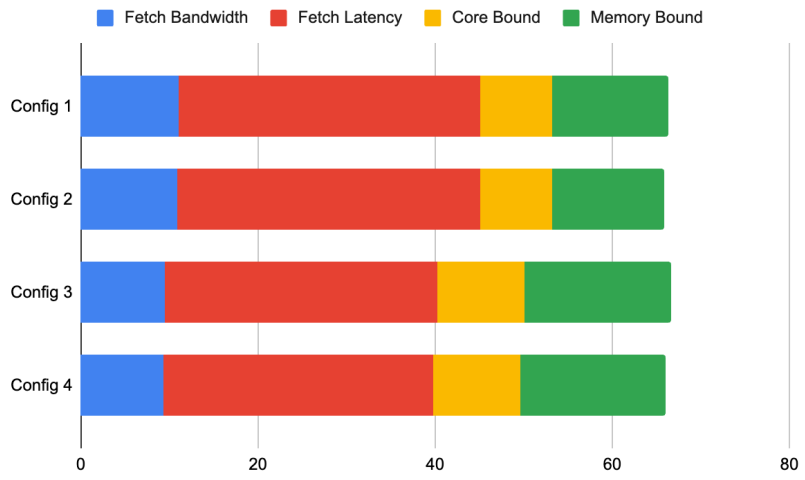

Fig. 8: frontend and backend perf metrics on different cache configurations

7. Results

This work provides a first look at how the structure of a simulation impacts the performance of gem5 simulations. Profiling with the top-down microarchitectural analysis process revealed that across configurations, gem5 remained heavily frontend bound, but the ISA and cache configuration could both have a large impact on the extent. In general, the RISC-V ISA performed the best, particularly on a more complex system configuration, where 32.84% of processor cycles were categorized as retiring, compared to 31.12% on x86 and 30.06% on ARM. The ARM ISA runs the least efficiently of the three, likely due in part to the larger amount of generated code that it relies on for decoding and executing instructions.

Experiments also showed ways to lower the runtime of experiments. The easiest means to achieve this was to lower the amount of cache misses, as gem5's DRAM model caused the simulation to run a significant amount of extra instructions compared to cache hits. Larger caches could speed up simulations by up to 39%, and more work needs to be done to examine how the working set of the program running in the simulation impacts this. Experiments on the workload of the simulation revealed similar results, showing that the fluidanimate benchmark, which is described as having a large working set, ran substantially slower than others, despite very little difference in the efficiency of the simulation itself. Increasing cache size provides a clear method to speed up simulations on experiments where detailed cache data is less important, and could be unintuitive to newer users.

Overall, this was a preliminary study, designed to look broadly at how gem5 runs, and what bottlenecks it faces on different simulations. Time constraints limited the amount of data that could be collected, and more work needs to be done to reinforce the results of this paper. Avenues for future studies will be discussed in the next section.

7.1 Future Work

Continuation of Current Work

Limited trials were performed for most of the experiments in this paper, and a higher number of trials would allow for more sound conclusions to be drawn. The amount of time that each individual experiment took to complete caused significant time constraints, and the amount data reflects these.

Beyond just running more of the same trials, the data in this paper has revealed multiple further trends to explore. First, more diverse experiments should be conducted on the ISA system to determine what aspects of the configuration impact each one in what way. The processor model was the largest change between the simple and complex systems in this paper's experiments, but the other details play a role as well.

For workloads, trials could be run on larger input sizes to see if the lack of performance differences came primarily from the overall size of gem5 overshadowing the characteristics of each benchmark. The cache experiments performed in this paper provided a clear avenue for speeding up simulations, but only in a general sense. Detailed experiments should be conducted on different configurations and workloads with known working sets to see how directly the runtime is impacted by the percentage of cache hits and misses. Additionally, more data should be gathered on systems with the same overall size of the cache, but organized in different levels. This would allow for differences in caches to not be overshadowed by differences in number of memory accesses, as the hit ratio would be the same.

Finally, there are many more aspects of a gem5 simulation than were experimented on in this paper. Two obvious other parameters to test are the number of cores in the system, and the size of memory.

Comparison of Profiling Data Between Different Simulators

Looking beyond gem5, there are many simulators with similar capabilities and characteristics, even if none are quite as versatile. Experiments could be performed on many of

these simulators modeling roughly the same system to compare profiling data, and look at how the more efficient simulators are able to achieve that. This could provide insight on how different simulation techniques perform, and potentially ways to improve upon existing practices.

Optimization Experiments on gem5

It's always difficult to improve on a codebase as large and well constructed as gem5's, but there are a few areas revealed by this paper that could be candidates for optimization. The ISA system is one clear target, as it relies on generated code to decode and execute instructions. Section 4.2 mentioned the idea of an analysis of instruction usage. The intention would be to track which instructions are executed inside the simulation, and where in memory they're fetched from. If data is gathered from diverse enough workloads, this could indicate how well the ISA system takes advantage of locality, and potentially provide an area for improvement based on instruction and micro-operation arrangement within the files.

References

[1] T. N. Theis and H.-S. P. Wong, “The end of moore’s law: A new beginning for information technology,” *Comput. Sci. Eng.*, vol. 19, no. 2, pp. 41–50, Mar. 2017.

[2] G. E. Moore, “Cramming more components onto integrated circuits,” *Proc. IEEE Inst. Electr. Electron. Eng.*, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[3] Association for Computing Machinery (ACM), “John Hennessy and David Patterson 2017 ACM A.M. Turing Award Lecture.” Accessed: May 03, 2025. [Online]. Available: https://www.youtube.com/watch?v=3LVeEjsn8Ts

[4] C. E. Leiserson *et al.*, “There’s plenty of room at the Top: What will drive computer performance after Moore's law?,” *Science*, vol. 368, no. 6495, p. eaam9744, Jun. 2020.

[5] J. Lowe-Power *et al.*, “The gem5 Simulator: Version 20.0+,” *arXiv [cs.AR]*, Jul. 06, 2020. [Online]. Available: http://arxiv.org/abs/2007.03152

[6] A. Kivity and A. Liguori, “kvm : the Linux Virtual Machine Monitor,” 2007, Accessed: May 03, 2025. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf

[7] M. M. K. Martin *et al.*, “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” *Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[8] A. Akram and L. Sawalha, “A survey of computer architecture simulation techniques and tools,” *IEEE Access*, vol. 7, pp. 78120–78145, 2019.

[9] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSSx86: A full system simulator for x86 CPUs,” Jan. 2011, Accessed: May 03, 2025. [Online]. Available: http://dx.doi.org/

[10] M. T. Yourst, “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator,” in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, IEEE, Apr. 2007, pp. 23–34.

[11] D. Sánchez and C. Kozyrakis, *ZSim: fast and accurate microarchitectural simulation of thousand-core systems*. 2013.

[12] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2Sim: A simulation framework for CPU-GPU computing,” in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, Sep. 2012, pp. 335–344.

[13] W. Heirman, T. E. Carlson, and L. Eeckhout, “Sniper: scalable and accurate parallel multi-core simulation,” pp. 91–94, 2012.

[14] “perf: Linux profiling with performance counters.” Accessed: May 03, 2025. [Online]. Available: https://perfwiki.github.io/main/

[15] “Intel® VTune™ Profiler,” Intel. Accessed: May 03, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.m17bbn

[16] “AMD μProf,” AMD. Accessed: May 03, 2025. [Online]. Available: https://www.amd.com/en/developer/uprof.html

[17] T.-Y. Liu, J. Guo, and B. Huang, “Efficient cross-platform multiplexing of hardware performance counters via adaptive grouping,” *ACM Trans. Archit. Code Optim.*, Oct. 2023, doi: 10.1145/3629525.

[18] A. Yasin, “A Top-Down method for performance analysis and counters architecture,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Mar. 2014, pp. 35–44.

[19] J. Umeike, N. Patel, A. Manley, A. Mamandipoor, H. Yun, and M. Alian, “Profiling gem5 Simulator,” in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Apr. 2023, pp. 103–113.
[20] C. Bienia, S. Kumar, J. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” *Int Conf Parallel Archit Compil Tech*, pp. 72–81, Oct. 2008.
[21] A. Akram and L. Sawalha, “A Comparison of x86 Computer Architecture Simulators,” 2016, Accessed: May 03, 2025. [Online]. Available: https://scholarworks.wmich.edu/casrl_reports/1/
[22] I. Hwang, J. Lee, H. Kang, G. Lee, and H. Kim, “Survey of CPU and memory simulators in computer architecture: A comprehensive analysis including compiler integration and emerging technology applications,” *Simul. Model. Pract. Theory*, vol. 138, no. 103032, p. 103032, Jan. 2025.
[23] W. Stallings, *Computer organization and architecture, global edition*, 11th ed. London, England: Pearson Education, 2023.
[24] A. B. Tucker and M. J. Flynn, “Dynamic microprogramming: processor organization and programming,” *Commun. ACM*, vol. 14, no. 4, pp. 240–250, Apr. 1971.
[25] “gem5: ISA Parser.” Accessed: May 09, 2025. [Online]. Available: https://www.gem5.org/documentation/general_docs/architecture_support/isa_parser/
[26] “os — Miscellaneous operating system interfaces,” Python documentation. Accessed: May 06, 2025. [Online]. Available: https://docs.python.org/3/library/os.html
[27] “taskset(1) - Linux manual page.” Accessed: May 06, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html
[28] “Documentation for /proc/sys/vm/ — The Linux Kernel documentation.” Accessed: May 03, 2025. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html#drop-caches
[29] “nohup(1p) - Linux manual page.” Accessed: May 06, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/nohup.1p.html
[30] B. Bruce *et al.*, “Enabling Reproducible and Agile Full-System Simulation: Artifact,” Mar. 07, 2021, *figshare*. doi: 10.6084/M9.FIGSHARE.14176802.
[31] “proc_pid_status(5) - Linux manual page.” Accessed: May 06, 2025. [Online]. Available: https://man7.org/linux/man-pages/man5/proc_pid_status.5.html
[32] “ps(1) - Linux manual page.” Accessed: May 06, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/ps.1.html

Appendix A: Abnormal Core Behavior on Server

When running the first batches of experiments, certain trials gave significantly different data than the rest. These did not act like normal outliers, as these abnormal trials were incredibly consistent in how they differed. These trials both took substantially longer, and were substantially more frontend bound. Table IX shows 10 early trials, with trials 5, 6, and 8 showing vastly different profiling data than the rest. Normally, some outliers are expected when performing experiments, but not to this extent or consistency. Investigations were performed to attempt to find any consistency between these trials. At three points during execution, information about the process from `/proc/{pid}/status` [31] was printed as well as what core the process most recently ran on using the Linux `ps` command [32]. After running more trials, processes running on cores 10 and 30 were consistently the ones giving these questionable results. This was further reinforced by the fact that core 30 is actually the hyperthread associated with core 10, meaning that they share the same cache, which could be the reason that both performed so much worse in the frontend bound category. Nothing was found to indicate that this core is intentionally different in some way, so it was assumed that this was caused by some kind of unknown issue. In order to avoid letting this core impact results, all further trials ran the `taskset` command [27] to set the core affinity of the process to every core except 10 and 30.

Table IX: 10 trials with outliers

Metric	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
tma_backend_bound	7.8	8.1	8.2	8.8	4.6	4.8	8.3	5.2	8.3	8.1
tma_bad_speculation	2.7	2.7	2.6	2.8	1.7	1.8	2.6	1.9	2.7	2.8
tma_frontend_bound	43.4	43.3	43.4	43.1	58.6	58	43.4	55.5	43.4	43.7
tma_retiring	46.2	45.9	45.8	45.3	35	35.4	45.7	37.4	45.7	45.5

Appendix B: gem5 Warnings and Bugs

Gem5 does an excellent job of modeling a CPU as closely as possible, but no system is perfect, and some issues were found over the course of these experiments. The occasional warning during execution is expected, after all, even smaller workloads are typically simulating tens of thousands to millions of instructions. One consistent example was the occasional obscure x86 instructions which weren't implemented, but in most cases the simulator could move past this without issue. The only significantly disruptive bug came from trying to run the MinorCPU model on x86 systems booting Ubuntu 18.04 (the version needed for PARSEC). At some point during execution, the simulator would spit out "`panic: Invalid microop!`" and quit immediately. Others have reported the same issue online, and it seems to be an ongoing problem without a clear fix. This halted plans for experiments on gem5's CPU models, as the MinorCPU would have provided a clear middle ground between the SimpleCPU and O3CPU.

Appendix C: Gem5 Script

```python
import argparse
import time
import os
import subprocess

import m5
from m5.objects import Root

import argparse

from gem5.coherence_protocol import CoherenceProtocol
from gem5.components.boards.x86_board import X86Board
from gem5.components.boards.arm_board import ArmBoard
from gem5.components.boards.riscv_board import RiscvBoard
from gem5.components.memory import DualChannelDDR4_2400
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.components.processors.simple_switchable_processor import
SimpleSwitchableProcessor
from gem5.components.cachehierarchies.ruby.mesi_two_level_cache_hierarchy
import MESITwoLevelCacheHierarchy
from gem5.components.cachehierarchies.ruby.mesi_three_level_cache_hierarchy
import MESIThreeLevelCacheHierarchy
from gem5.isas import ISA
from gem5.resources.resource import obtain_resource
from gem5.simulate.exit_event import ExitEvent
from gem5.simulate.simulator import Simulator
from gem5.utils.requires import requires

isa_choices = [
```

```
    "X86",
    "ARM",
    "RISCV",
]

cpu_choices = [
    "TIMING",
    "MINOR",
    "O3",
]

workload_choices = [
    "boot-exit",
    "blackscholes",
    "dedup",
    "fluidanimate",
    "canneal",
]

parser = argparse.ArgumentParser()

parser.add_argument(
    "--isa",
    type=str,
    required=True,
    choices=isa_choices,
)

parser.add_argument(
    "--cpu",
    type=str,
    required=True,
    choices=cpu_choices,
)
```

```
parser.add_argument(
    "--mem",
    type=str,
    required=True,
)

parser.add_argument(
    "--cache",
    type=str,
    required=True,
)

parser.add_argument(
    "--cores",
    type=int,
    required=True,
)

parser.add_argument(
    "--workload",
    type=str,
    required=True,
    choices=workload_choices,
)

args = parser.parse_args()

sizes = args.cache.split(":")

if (len(sizes) == 2):
    cache_hierarchy = MESITwoLevelCacheHierarchy(
        l1d_size=sizes[0],
        l1d_assoc=8,
```

```
        l1i_size=sizes[0],
        l1i_assoc=8,
        l2_size=sizes[1],
        l2_assoc=16,
        num_l2_banks=2,
    )
elif (len(sizes) == 3):
    cache_hierarchy = MESIThreeLevelCacheHierarchy(
        l1d_size=sizes[0],
        l1d_assoc=8,
        l1i_size=sizes[0],
        l1i_assoc=8,
        l2_size=sizes[1],
        l2_assoc=12,
        l3_size=sizes[2],
        l3_assoc=16,
        num_l3_banks=2,
    )
else:
    print("error in cache description")
    exit(1)

memory = DualChannelDDR4_2400(size=args.mem)

if (args.workload == "boot-exit"):
    processor = SimpleProcessor(
        cpu_type=CPUTypes[args.cpu],
        isa=ISA[args.isa],
        num_cores=args.cores,
    )

    if (args.isa == "X86"):
        board = X86Board(
            clk_freq="3GHz",
```

```
            processor=processor,
            memory=memory,
            cache_hierarchy=cache_hierarchy
        )
        workload = obtain_resource("x86-ubuntu-24.04-boot-no-systemd")
        board.set_workload(workload)
    elif (args.isa == "RISCV"):
        board = RiscvBoard(
            clk_freq="3GHz",
            processor=processor,
            memory=memory,
            cache_hierarchy=cache_hierarchy,
        )
        workload = obtain_resource("riscv-ubuntu-24.04-boot-no-systemd")
        board.set_workload(workload)
    elif (args.isa == "ARM"):
        board = ArmBoard(
            clk_freq="3GHz",
            processor=processor,
            memory=memory,
            cache_hierarchy=cache_hierarchy
        )
        workload = obtain_resource("arm-ubuntu-24.04-boot-no-systemd")
        board.set_workload(workload)
    else:
        print("error in isa description")
        exit(1)
    def exit_event_handler():
        print("first exit event: kernel booted")
        print(time.strftime("%Y-%m-%d-%H-%M-%S"))
        yield False

        print("second exit event: in after boot")
        print(time.strftime("%Y-%m-%d-%H-%M-%S"))
```

```
        yield False

        print("third exit event: after run script")
        print(time.strftime("%Y-%m-%d-%H-%M-%S"))
        yield True

    simulator = Simulator(
        board = board,
        on_exit_event = {
            ExitEvent.EXIT: exit_event_handler(),
        },
    )
else:

    processor = SimpleSwitchableProcessor(
        starting_core_type=CPUTypes.KVM,
        switch_core_type=CPUTypes[args.cpu],
        isa=ISA[args.isa],
        num_cores=args.cores,
    )

    command = (
        f"cd /home/gem5/parsec-benchmark;"
        + "source env.sh;"
        + f"parsecmgmt -a run -p {args.workload} -c gcc-hooks -i simsmall
-n {args.cores};"
        + "sleep 5;"
        + "m5 exit;"
    )
    board = X86Board(
        clk_freq="3GHz",
        processor=processor,
        memory=memory,
        cache_hierarchy=cache_hierarchy
```

```
)
board.set_kernel_disk_workload(
    kernel=obtain_resource(
        "x86-linux-kernel-4.19.83", resource_version="1.0.0"
    ),
    disk_image=obtain_resource("x86-parsec", resource_version="1.0.0"),
    readfile_contents=command,
)
def handle_workbegin():
    print("Done booting Linux")
    processor.switch()
    print(time.strftime("%Y-%m-%d-%H-%M-%S"))
    print("ps: ")
    s = "ps -o pid,psr,comm -p " + str(os.getpid())
    result = subprocess.run(s, shell=True, capture_output=True, text=True)

    if result.returncode == 0:
        print("success")
        print(result.stdout)
    else:
        print("error executing command")
        print(result.stderr)
        exit(1)
    yield False

def handle_workend():
    print("end of benchmark, stat dump")
    m5.stats.dump()
    print()
    print(time.strftime("%Y-%m-%d-%H-%M-%S"))

simulator = Simulator(
    board=board,
    on_exit_event={
```

```
            ExitEvent.WORKBEGIN: handle_workbegin(),
            ExitEvent.WORKEND: handle_workend(),
        },
    )

simulator.run()
```